Java Swing Custom GUI MVC Component Tutorial

Prakash Manandhar, Sagun Dhakhwa, Ishwor Thapa
Sambad Project (www.sambad.org), Madan Puraskar Pustakalaya, Nepal

July 13, 2006

1 Introduction

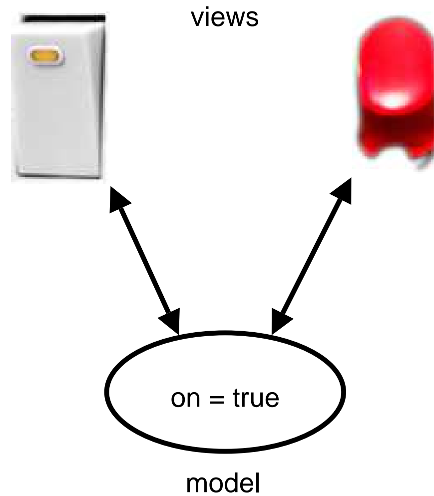
The Model-View-Controller (MVC) architecture is one of the first widely acclaimed architectures for GUI systems (Gamma et al., 1995;Buschmann et al., 1996). Contemporary GUI APIs (and even some Web APIs) - like Java Swing, Microsoft Foundation Classes and even its newest incarnation WinFX - are also based on modifications of the MVC architectural pattern (Eckstein et al., 1998; Kruglinski, 1995; Livermore et al., 2006). The MVC does have its fair share of critics, for example Holub [1999] goes so far as to say that MVC is *not* object-oriented, but we think it is a logical and proven system for beginning GUI system designers and programmers.

In this tutorial we take you through creating several simple custom components from scratch using Java Swing as the implementation platform. It is hoped that readers will get a better understanding of the MVC architecture from this document as well as find within it a cook-book for creating MVC style custom GUI components for Java Swing.

We implement a simple GUI with toggle switches and LED displays that controls output to the console.

Figure 1: Screenshot of finished program

Figure 2: Model View Controller Architecture



2 Model View Controller

The Model View Controller Architecture (MVC) is software design architectural pattern which breaks an application or just an application's interface into three types of objects namely, Model, View and Controller (Gamma et al., 1995). Model is the application object, View is the screen presentation and Control determines how UI reacts to the user input. MVC separates these three components, which used to be coupled together in architectures prior to MVC, yielding more flexibility and reusability. The model encapsulates more than just data and function that operates on it. It represents not just the state of the UI but also how the UI works, making it easier to use real-world modeling (e.g,: Switch and Bulb) techniques in defining the models (Helman, 1998).The model contains the actual state of the UI component. When the state of the UI component changes, the model data should reflect the change. The model then notifies all the views which subscribe it, in response the views update themselves. View is responsible for displaying the state of the model on a device. It attaches to a model and render its contents on the device. The controller subscribes both the model and the viewport and knows the type of both in order to translate user input into application response.The views in a MVC can be nested, i.e. they can contain server sub-views.

The MVC allows to change the response of the view to the user input without changing the view presentation. A view uses the controller to implement a particual response strategy of the response.

3 Writing the Model

Toggle Model represents the state of the component. The model has either the 'ON' state or 'OFF' state. It also coordinates with the view by notifying when this model changes the state.

To represent the state of the model, we simply define a boolean variable called *on*, which when 'true' marks the 'ON' state of the model and 'false' marks the 'OFF' state of the model.

To maintain link between model and view, model should notify its views with a message that the state has been changed. Here in this example when the state is changed, the Toggle model notifies all its event listeners. Listeners are registered to this model by addToggleListener() call.

```
package np.org.mpp.sambad.tutorial.mvc;

import java.util.ArrayList;
import java.util.List;

/**
 * A simple model that keeps track whether something is On or Off.
 * @author Prakash Manandhar
 */
public class ToggleModel {

        // model state
        private boolean on;

        // list of listeners to this model
        private List<ToggleListener> listeners = new ArrayList<ToggleListener>();

        /**
         * Default constructors. Initially the model is <em>not on</em>.
         */
        public ToggleModel ()
        {
                on = false;
        }

        /**
         * Constructor with given intial state.
         * @param isOn
         */
        public ToggleModel (boolean isOn)
        {
                on = isOn;
        }

        /**
         * @return is this toggle on?
         */
        public boolean isOn ()
        {
                return on;
        }

        /**
         * Toggles isOn from true to false and vice versa. Also fires
         * toggleChanged event.
         */
        public void toggle ()
        {
                on = !on;
                fireEvent();
        }

        /**
```

```
         * Adds given listener to listener list.
         * @param l
         */
        public void addToggleListener (ToggleListener l)
        {
                listeners.add(l);
        }

        /**
         * Removes given listener from listener list.
         * @param l
         */
        public void removeToggleListener (ToggleListener l)
        {
                listeners.remove(l);
        }

        /**
         * Fires toggle event.
         */
        private void fireEvent ()
        {
                for (ToggleListener l: listeners)
                        l.toggleChanged(this);
        }
}
```

4 Writing the JComponent

To write a custom UI component in swing, one has to create a class extending JComponent which is a base class for both standard and custom components (Eckstein et al., 1998). In this tutorial, we have shown how to create ToggleSwitch and ToggleLED UI components which share a common model, ToggleModel. The job of component class which extends abstract JComponent is to tie together model and the UI delegate together. The custom component does this with the help of Constructor and setUI() method respectively. It also implements a listener interface (ToggleListener in our case) which has a stateChanged method (toggleChanged in our case) to notify a state change. The custom component changes in rendering on the viewport through the UI delegate, in accordance to the change in state notified by stateChanged method.

```
        public ToggleSwitch ()
        {
                init (new ToggleModel());
        }

        public ToggleSwitch (boolean isOn)
        {
                init (new ToggleModel(isOn));
        }


        public ToggleSwitch (ToggleModel model)
        {
                init (model);
        }
```

4.1 The toggle switch component

Toggle switch component simulates the real world ‘switch’ having two states ON / OFF. It changes the state of the LED display, changing its own state when user clicks on the switch, in similar manner as the real world “switch” changes the state of a bulb / LED when toggled.

There are three ways to instantiate a toggle switch component, allowing a programmer to instantiate it with a default ToggleModel (ToggleSwitch()), with a model with specified on/off state (ToggleSwitch(boolean isOn))and a specified instance of ToggleModel. All the three forms of constructors call the init(ToggleModel m) method.

```
        public ToggleSwitch ()
        {
                init (new ToggleModel());
        }

        public ToggleSwitch (boolean isOn)
        {
                init (new ToggleModel(isOn));
        }


        public ToggleSwitch (ToggleModel model)
        {
                init (model);
        }
```

The init inititalises the toggle switch component by setting the given model, registering the toggle switch component in the model as a listener to listen the toggle change during user input, initializing the preferred dimension, loading all the icons, setting the keyboard maps and finally setting up the UI property to current UI delegate by updateUI property.

```
private void init (ToggleModel model)
{
        this.model = model;
        model.addToggleListener(this);
        setKeyboardMaps();
        loadIcons();
        setPreferredSize(new Dimension(onIcon.getIconWidth(),
        updateUI();
}
```

The setKeyboardMaps method TAB as the forward traversal key and Shift + TAB as the reverse traversal key. It first creates an instance of HashSets called keystrokes and keystrokes_back, into which it adds keystrokes for TAB and Shift + TAB respectively. Then setFocusTraversalKeys method is used to add focus traversal keys by passing these traversal key type with the HashSet.

```
private void setKeyboardMaps() {
         Set<AWTKeyStroke> keystrokes = new HashSet<AWTKeyStroke>();
         keystrokes.add(AWTKeyStroke.getAWTKeyStroke("TAB"));
         setFocusTraversalKeys(KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS, keystrokes);
         Set<AWTKeyStroke> keystrokes_back = new HashSet<AWTKeyStroke>();
         keystrokes_back.add(AWTKeyStroke.getAWTKeyStroke("shift TAB"));
         setFocusTraversalKeys(KeyboardFocusManager.BACKWARD_TRAVERSAL_KEYS,
                                                                  keystrokes_back);
}
```

Here is the complete list of ToggleSwitch.java:

```
package np.org.mpp.sambad.tutorial.mvc;

import java.awt.AWTKeyStroke;
import java.awt.Dimension;
import java.awt.KeyboardFocusManager;
import java.util.HashSet;
import java.util.Set;

import javax.swing.ImageIcon;
import javax.swing.JComponent;
import javax.swing.UIManager;

/**
 * A simple toggle switch component.
 * @author Prakash Manandhar
 */
public class ToggleSwitch extends JComponent
implements ToggleListener
{

        /**
         * For serialization.
         */
        private static final long serialVersionUID = 8845972060032351155L;

        // model for this component
        private ToggleModel model;

        private ImageIcon onIcon;
        private ImageIcon offIcon;

        private static final String onIconPath = "res/toggle_on.jpg";
        private static final String offIconPath = "res/toggle_off.jpg";

        /**
         * Creates a default toggle switch with a default constructor.
         */
        public ToggleSwitch ()
        {
                init (new ToggleModel());
        }

        /**
         * Creates a toggle switch with a model with given initial state.
         * @param isOn
         */
        public ToggleSwitch (boolean isOn)
        {
                init (new ToggleModel(isOn));
        }

        /**
         * Creates a ToggleSwitch with given model.
         * @param model
         */
        public ToggleSwitch (ToggleModel model)
        {
                init (model);
        }

        /**
```

```
 * sets the model of this component and adds it to listener
 * list.
 * @param model
 */
private void init (ToggleModel model)
{
        this.model = model;
        model.addToggleListener(this);
        setKeyboardMaps();
        loadIcons();
        setPreferredSize(new Dimension(onIcon.getIconWidth(),
         onIcon.getIconHeight()));
        updateUI();
}

/**
 * Resets the UI property to the current UIdelegate for this
 * component.
 */
public void updateUI ()
{
        setUI((np.org.mpp.sambad.tutorial.mvc.ToggleSwitchUI)UIManager.getUI(this));
        invalidate();
}

/**
 * @return The UIDefaults key used to look up the name of
 * the ComponentUI class that defines the look and feel
 * for this component.
 */
public String getUIClassID() {
        return "np.org.mpp.sambad.tutorial.mvc.ToggleSwitch";
}

/**
 * Returns the model of this (ToggleSwitch) component
 * @return model.
 */
public ToggleModel getModel ()
{
        return model;
}

/**
 * Set TAB as Forward Traversal Key and
 * SHIFT + TAB as Backward Traversal Key
 *
 */

private void setKeyboardMaps() {
        Set<AWTKeyStroke> keystrokes = new HashSet<AWTKeyStroke>();
        keystrokes.add(AWTKeyStroke.getAWTKeyStroke("TAB"));
     setFocusTraversalKeys(KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS, keystrokes);
        Set<AWTKeyStroke> keystrokes_back = new HashSet<AWTKeyStroke>();
        keystrokes_back.add(AWTKeyStroke.getAWTKeyStroke("shift TAB"));
setFocusTraversalKeys(KeyboardFocusManager.BACKWARD_TRAVERSAL_KEYS, keystrokes_back);
}

/**
 * repaints the component
 */
public void toggleChanged(ToggleModel source) {
```

```
		repaint ();
	}

	/**
	 * sets onIcon and offIcon to their corresponding path
	 */
	private void loadIcons ()
	{
		onIcon = new ImageIcon(onIconPath);
		offIcon = new ImageIcon(offIconPath);
	}

	/**
	 * Returns Switch-ON image
	 * @return onIcon
	 */
	ImageIcon getOnIcon ()
	{
		return onIcon;
	}

	/**
	 * Return LED-OFF image
	 * @return offIcon
	 */
	ImageIcon getOffIcon ()
	{
		return offIcon;
	}

}
```

4.2 The LED display component

LED display component simulates the real world LED implementing ToggleListener as the toggle switch component does. They both implement ToggleListener interface because both of them displays the same state, i.e. ON / OFF state. Implementation of LED display component is very similar to that of toggle switch component . Here is the complete list of ToggleLED.java:

```
package np.org.mpp.sambad.tutorial.mvc;

import java.awt.AWTKeyStroke;
import java.awt.Dimension;
import java.awt.KeyboardFocusManager;
import java.util.HashSet;
import java.util.Set;

import javax.swing.ImageIcon;
import javax.swing.JComponent;
import javax.swing.UIManager;

/**
 * A simple toggle LED display component.
 * @author Prakash Manandhar
 */
public class ToggleLED extends JComponent
implements ToggleListener
{
```

```
/**
 * For serialization.
 */
private static final long serialVersionUID = 8845972060032351155L;

// model for this component
private ToggleModel model;

private ImageIcon onIcon;
private ImageIcon offIcon;

private static final String onIconPath = "res/led_on.jpg";
private static final String offIconPath = "res/led_off.jpg";

/**
 * Creates a default toggle switch with a default constructor.
 */
public ToggleLED ()
{
        init (new ToggleModel());
}

/**
 * Creates a toggle switch with a model with given initial state.
 * @param isOn
 */
public ToggleLED (boolean isOn)
{
        init (new ToggleModel(isOn));
}

/**
 * Creates a ToggleSwitch with given model.
 * @param model
 */
public ToggleLED (ToggleModel model)
{
        init (model);
}

/**
 * sets the model of this component and adds it to listener.
 * @param model
 */
private void init (ToggleModel model)
{
        this.model = model;
        model.addToggleListener(this);
        setKeyboardMaps();
        loadIcons();
        setPreferredSize(new Dimension(onIcon.getIconWidth(),
                                                                        onIcon.getIconHeig
        updateUI();
}
/**
 * Resets the UI property to the current UIdelegate for this
 * component.
 */
public void updateUI ()
{
        setUI((np.org.mpp.sambad.tutorial.mvc.ToggleLEDUI)UIManager.getUI(this));
        invalidate();
```

```
}
/**
 * @return The UIDefaults key used to look up the name of
 * the swing.plaf.ComponentUI  class that defines the look
 * and feel for this component.
 */
public String getUIClassID() {
        return "np.org.mpp.sambad.tutorial.mvc.ToggleLED";
}

/**
 * Returns the model of this (ToggleLED) component
 * @return model.
 */
public ToggleModel getModel ()
{
        return model;
}

/**
 * Set TAB as Forward Traversal Key and
 * SHIFT + TAB as Backward Traversal Key
 *
 */
private void setKeyboardMaps() {
        Set<AWTKeyStroke> keystrokes = new HashSet<AWTKeyStroke>();
        keystrokes.add(AWTKeyStroke.getAWTKeyStroke("TAB"));
        setFocusTraversalKeys(KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS, keystrokes);
        Set<AWTKeyStroke> keystrokes_back = new HashSet<AWTKeyStroke>();
        keystrokes_back.add(AWTKeyStroke.getAWTKeyStroke("shift TAB"));
        setFocusTraversalKeys(KeyboardFocusManager.BACKWARD_TRAVERSAL_KEYS, keystrokes_bac
}

/**
 * repaints the component
 */
public void toggleChanged(ToggleModel source) {
        repaint ();
}

/**
 * sets onIcon and offIcon to their corresponding path
 */
private void loadIcons ()
{
        onIcon = new ImageIcon(onIconPath);
        offIcon = new ImageIcon(offIconPath);
}

/**
 * Returns LED-ON image
 * @return onIcon
 */
ImageIcon getOnIcon ()
{
        return onIcon;
}

/**
 * Return LED-OFF image
 * @return offIcon
 */
```

```
        ImageIcon getOffIcon ()
        {
                return offIcon;
        }

}
```

5 Writing the UI Delegate

A UI Delegate is a composition of view and controller bundled together. As to our example, we have two UI delegates namely, ToggleLedUI and ToggleSwitchUI. UI delegates should have the createUI() method defined that returns the ComponentUI / UI delegate.

Since ToggleLedUI does not entertain any user input directly, it has only the view part, which is used to render (or show) the component on the screen by calling the paint() method. However, the ToggleSwitchUI has both the controller and the view parts, because it has to listen to events associated to it.

View layer of both of these components have simply the paint() method that renders each of them based on the same underlying model called Toggle Model. The method paint() creates respective component, define the component's model, and draws the component's image.

Controller layer in ToggleSwitchUI implements mouse listener and respond to mouseClick event on the ToggleSwitch component. For this, event listeners are to be installed onto the component done by installUI() method.

paint() method for ToggleLedUI delegate:

```
        public void paint (Graphics g, JComponent c)
        {
                ToggleLED ts  = (ToggleLED)c;
                ToggleModel model = ts.getModel();
                Dimension size = ts.getSize();
                if (model.isOn())
                        drawIcon (g, ts.getOnIcon(), size);
                else
                        drawIcon (g, ts.getOffIcon(), size);
        }
```

installUI() and paint() method for ToggleSwitchUI delegate:

```
        /**
         * Configures the specified component and creates
         * event listener on the component.
         *
         * @param c
         */
        public void installUI (JComponent c)
        {
                c.addMouseListener(controller);
        }

        /**
         * Paints the specified component.
         */
```

```
	public void paint (Graphics g, JComponent c)
	{
		ToggleSwitch ts  = (ToggleSwitch)c;
		ToggleModel model = ts.getModel();
		Dimension size = ts.getSize();
		if (model.isOn())
			drawIcon (g, ts.getOnIcon(), size);
		else
			drawIcon (g, ts.getOffIcon(), size);
	}
```

6 Linking to the model

We have already explained that a single model can be used by different views. But it doesn't necessarily mean that the view only needs model. Here we describe how system's console output can be linked to the same ToggleModel. With respect to the state of the model, the console output is being generated so that it acts as a model listener. Just as we have made the console output to be the model listener of our model, we can make other application oriented listeners that listen to the same model. For an example, we can add in a parallel port interfacing class as another listener to this model and instead of displaying output (states) onto the console, the class sending the output through the parallel port.

BitOutputConsole class is the model listener to the Toggle Model. And a link to the model is created by the addModel() function where BitOutputConsole (our model listener) registers itself as a listener to the model. It also implements the ToggleListener class and on doing so, it defines the toggleChanged() method which is called upon by model whenever the state is changed. The method toggleChanged() displays the state of all the models registered in the list of models for the BitOutputConsole .

```
package np.org.mpp.sambad.tutorial.mvc.test;

import java.util.ArrayList;
import java.util.List;

import np.org.mpp.sambad.tutorial.mvc.ToggleListener;
import np.org.mpp.sambad.tutorial.mvc.ToggleModel;

/**
 * Demonstration of ModelListener.
 * @author Prakash Manandhar
 */
public class BitOutputConsole implements ToggleListener {

	List<ToggleModel> models = new ArrayList<ToggleModel>();

	/**
	 * Adds Model to the list of models for this listener and
	 * adds itself to the listener's list of the model.
	 * @param model
	 */

	public void addModel (ToggleModel model)
	{
		models.add (model);
		model.addToggleListener(this);
	}
```

```
        /**
         * Writes the state of all LEDs in console.
         * LED ON State = 1, LED OFF State = 0
         */

        public void toggleChanged(ToggleModel source) {
                System.out.println("");
                for (ToggleModel m: models)
                        System.out.print(m.isOn()?1:0);
        }

}
```

7 The main class

The main class in the ToggleComponentTest initializes the UIManager by adding ToggleSwitchUI and ToggleLEDUI, which will be used by the custom components to set the UI delegate. After that a parent container for the custom components, JFrame is initialized to whose content pane, 8 pairs of toggle switch and toggle LED sharing a separate toggle model is added using flowlayout, layout manager. Each pair of toggle switch and toggle LED sharing a toggle model is instantiated with the help of static method called addToggleSet. The addToggleSet method takes an integer to calculate ON / OFF state of the model, a container to which the toggle switch and toggle LED component pairs are to be added and bit output console, which displays the state bit of the model on the console. It creates a box container, to which the switch and LED pair is added, which in turn is added to the container. Finally the main class packs the added boxes, sets the default close operation and makes the frame visible.

```
public static void main(String[] args) {
                initUIManager();
                JFrame frame = new JFrame("www.sambad.org MVC Custom Component Tutorial");
                frame.getContentPane().setBackground(Color.WHITE);
                frame.getContentPane().setLayout(new FlowLayout());
                BitOutputConsole bo = new BitOutputConsole();
                for (int i = 0; i < 8; i++)
                        addToggleSet (i, frame.getContentPane(), bo);
                frame.pack();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
}
```

Here is the complete listing of ToggleComponentTest.java.

```
package np.org.mpp.sambad.tutorial.mvc.test;

import java.util.ArrayList;
import java.util.List;

import np.org.mpp.sambad.tutorial.mvc.ToggleListener;
import np.org.mpp.sambad.tutorial.mvc.ToggleModel;

/**
 * Demonstration of ModelListener.
 * @author Prakash Manandhar
 */
```

```
public class BitOutputConsole implements ToggleListener {

        List<ToggleModel> models = new ArrayList<ToggleModel>();

        /**
         * Adds Model to the list of models
         * @param model
         */

        public void addModel (ToggleModel model)
        {
                models.add (model);
                model.addToggleListener(this);
        }


        /**
         * writes the state of all LEDs in console.
         * LED ON State = 1, LED OFF State = 0
         */

        public void toggleChanged(ToggleModel source) {
                System.out.println("");
                for (ToggleModel m: models)
                        System.out.print(m.isOn()?1:0);
        }

}
```

Here is the complete listing of BitOutputConsole.java.

8 Summary

It is easy to write new Java Swing Components from scratch that follow the MVC architecture by following a few simple steps:

1. Create the model (reuse existing model if possible)
2. Create the JComponent, listen to the model and repaint when model changes
3. Create the ComponentUI, paint based on the state of the model
4. Create the Controller, listen to mouse and keyboard events; change the model when these events occur.

References

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns*. John Wiley & Sons, 1996.

Robert Eckstein, Marc Loy, and Dave Wood. *Java Swing*. O'Reilly, 1998.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns – elements of reusable object-oriented software*. Addison Wesley, 1995.

Dean Helman. Model-view-controller. Web, May 1998.

Allen Holub. Building user interfaces for object-oriented systems, part 1 – what is an object? the theory behind object-oriented user interfaces. Internet Journal/Magazine, July 1999.

D. Kruglinski. *Inside Visual C++*. Microsoft Press, 1995.

Shawn Livermore, Chris Andrade, and Scott Van Vliet. *Professional WPF Programming: .NET Development with the Windows Presentation Foundation*. Wiley, 2006.